VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Kompresní algoritmus LZ77 na GPU
Compression Algorithm LZ77 on GPU

2012 Petr Hlavinka

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Petr Hlavinka**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Kompresní algoritmus LZ77 na GPU
Compression Algorithm LZ77 on GPU

Zásady pro vypracování:

Cílem práce bude vyzkoušet implementaci kompresního algoritmu LZ77 prostřednictvím technologie CUDA/OpenCL na grafické kartě. Výsledkem by měla být funkční implementace na CPU a GPU, které budou porovnány z hlediska efektivity.

Obsah práce:
1. Prozkoumání technologie CUDA/OpenCL.
2. Popis kompresní algoritmu LZ77.
3. Návrh implementace algoritmu na GPU - definice optimalizací.
4. Implementace GPU algoritmu a referenční CPU implementace.
5. Porovnání CPU a GPU verze algoritmu.

Seznam doporučené odborné literatury:

Data Compression: The Complete Reference, David Salomon, 4. ed., Springer, 2007
CUDA Reference Manual

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Jan Platoš, Ph.D.**

Datum zadání: 18.11.2011
Datum odevzdání: 04.05.2012

VYSOKÁ ŠKOLA BÁŇSKÁ TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky OSTRAVA

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.*

V Ostravě 17. dubna 2012 .....

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 17. dubna 2012 .....

Rád bych poděkoval panu Ing. Janovi Platošovi Ph.D. za odbornou pomoc a konzultaci při vytváření této práce. Jen díky jeho pomoci a radám jsem byl schopen tuto bakalářskou práci dokončit v čtivé podobě.

Abstrakt

Smyslem této práce je implementace slovníkového kompresního algoritmus zvaného LZ77 a jeho akcelerace na grafické kartě. Na začátku bude vysvětleno jak algoritmus LZ77 funguje. S tím bude souviset i zamyšlení se nad implementací a optimalizací pro CPU a GPU. V první fázi bude ukázána implementace LZ77 pro procesor, další fáze se bude věnovat implementaci programu pro grafickou kartu s využitím frameworku CUDA. V závěru budou obě verze srovnány z hlediska efektivity, rychlosti a optimalizace.

Klíčová slova

Komprese, LZ77, CUDA, GPU, slovníková metoda komprese

Abstract

The purpose of this work is to implement a dictionary compression algorithm called LZ77 and its acceleration on the graphics card. At the beginning I will explain how algorithm LZZ works. With that will be related and reflect on the implementation and optimization for the CPU and GPU. In the first phase will be shown LZ77 implementation for the processor, the next phase will focus on program implementation for the video card using the CUDA framework. In conclusion, both versions will be compared in terms of efficiency, speed and optimization.

Key words

Compression, LZ77, CUDA, GPU, dictionary compression methods

Seznamy obrázků

Seznam použitých symbol a zkratek

CPU	Central processing unit
CUDA	Compute Unified Device Architecture
FLAC	Free Lossless Audio Codec
GPS	Global positioning system (globální systém určení polohy)
GPU	Graphics processing unit
JPEG	JPEG File Interchange Format
MP3	Motion Picture experts group - layer 3
LZ77	Lempel-Ziv 77 (bezeztrátový slovníkový kompresní algoritmus)

Obsah

1. Úvod

V dnešní době je všude kolem nás spousta elektronických zařízení. Ať se jedná o počítače, GPS navigace, mobilní telefony či třeba televize je třeba vždy zpracovávat nějaké informace. Pokud jsou zpracované informace nějakým způsobem důležité, je vhodné tyto informace uložit, abychom s nimi mohli v budoucnu pracovat. Každé zařízení má svůj vymezený diskový prostor pro uložení těchto dat. Cena za diskové úložiště již v dnešní době není nijak vysoká, avšak při intenzivní práci s počítačem není problémem během několika měsíců nastřádat velké množství dat. Komprese dat je metoda, která využívá různých algoritmů k zmenšení souborů, tím pádem se zmenší nárok na potřebu zdrojů pro uložení.

Rozlišujeme dva způsoby komprese dat. Bezeztrátovou a ztrátovou. Bezeztrátová metoda komprese je metoda, která nám dovoluje rekonstruovat data přesně do stavu před kompresí, takže nepřicházíme o žádnou informaci. Příkladem bezeztrátové komprese je FLAC pro audio, či LZ77 pro text. Ztrátová komprese je metoda, při níž jsou některá méně potřebná data vypouštěna, takže již nikdy nelze sestavit data původní. Po přečtení člověka hned napadne, jaká data můžou být v informatice méně potřebná. Využívá se jednoduše nedokonalosti lidských smyslů.

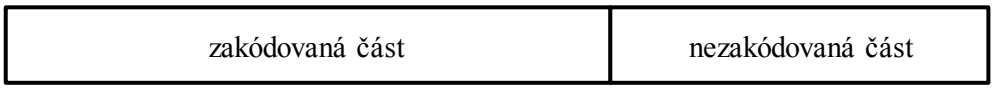
Obrázek uložen v bezeztrátovém formátu BMP s rozlišením 1366*768 pixelů potřebuje na uložení 1366*768*24/8 bajtů. Lidské oko ale není tak dokonalé, aby rozeznalo každý pixel, a proto se provede několik analýz jako je například podvzorkování, které sníží přesnost informace o barvě. Díky tomu je pak velikost obrázku JPEG cca. 5 krát menší než BMP, záleží však na typu obrázku. U MP3 komprese je zase využito nedokonalosti lidského sluchu. Pokud v záznamu hraje bubeník s kytaristou a bubeník silně udeří do bubnu, lidské ucho na chvíli neslyší frekvence kytary a právě tyto frekvence jsou u MP3 komprese vypouštěny. Jednoduše řečeno jsou odebrány informace, které člověk většinou neslyší.

Kompresní poměr lze velmi jednoduše spočítat pomocí následujícího vzorce:

$$kompresní\ poměr = \frac{velikost\ zkomprimovaného\ souboru}{velikost\ nezkomprimovaného\ souboru}$$

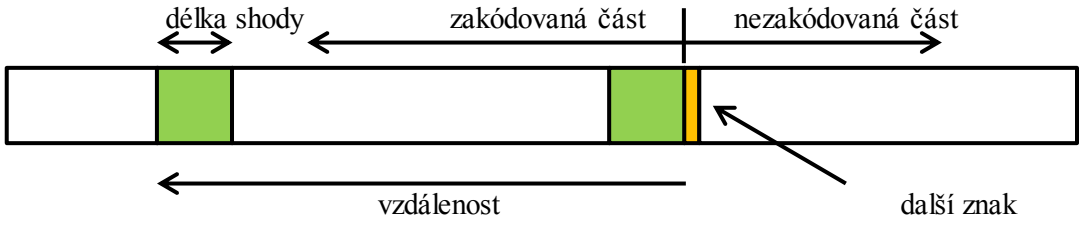
2. Popis kompresní metody LZ77

LZ77 je kompresní algoritmus vymyšlený Abrahamem Lempelem a Jacobem Zivem roku 1977. Jedná se o slovníkový algoritmus schopný komprimovat text. Slovníkový algoritmus hledá v textu redundantní data, která následně zakóduje. LZ77 využívá ke kódování metodu zvanou „sliding window", neboli posuvné okénko. Jedná se o pole, které se posunuje přes nekódovaný text. Jako slovník, v kterém LZ77 vyhledává redundance, se využívá již zakódovaná část posuvného okénka [1].



Obrázek 2.1 Posuvné okénko

V zakódované části jsou uchovány znaky, které již byly algoritmem zpracovány. Důležitým prvkem algoritmu je ukazatel pozice, který ukazuje na *M+1* znak, neboli na první nezakódovaný znak. Na začátku algoritmu se v posuvném okénku nenachází žádná data. Zakódování dat spočívá ve vytvoření trojice *(vzdálenost, délka, znak)*. Vzdálenosti je číselná hodnota, která značí vzdálenost ve znacích, v jaké se nachází shoda vzhledem k aktuální pozici algoritmu v textu. Délka je taktéž číselná hodnota, která značí počet shodných znaků. Není-li nalezen v zakódované částí první znak z části nezakódované je zapsána trojice *0,0,nenalezený znak*. Je-li nalezena shoda, algoritmus zakóduje se sekvenci (*vzdálenost, délka, první neshodný znak z nezakódované části*). Níže v této práci bude ukázáno několik způsobů pro samotné vyhledávání shod v textu.



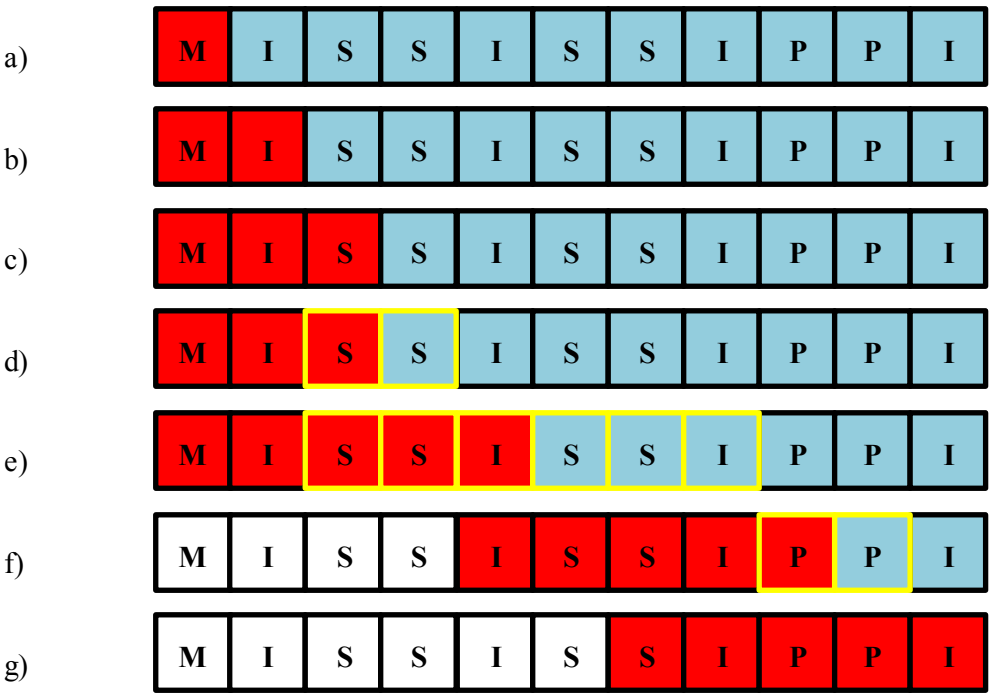
Obrázek 2.2 Nalezení shody v posuvném okénku

V algoritmu však existuje jedna výjimka. Objeví-li se v nezakódované části sekvence ABABABZ a v zakódované části je poslední zakódovaná sekvence AB, postupuje algoritmus následovně. Zakóduje sekvenci znaků ABABABZ jako trojici 2,6,Z, což značí, že v nezakódované části existuje vzhledem k aktuální pozici okénka v textu shoda délky šesti znaků AB. Po zakódování této trojice se algoritmus posouvá o 6 znaků vpřed.

Obrázek 2.3 Speciální případ kódování

2.1. Kódování LZ77

Představme si text „MISSISSIPPI“, který bychom chtěli zakódovat metodou LZ77, s velikostí posuvného okénka 5 znaků. Světle modrou barvou jsou vyznačena data, která ještě nebyla zakódována, červenou barvou je označena zakódovaná část, v které algoritmus vyhledává, žlutou barvou je vyznačena nejdelší shoda znaků. Důležité je si uvědomit, že pokud algoritmus najde shodu délky N, pokusí se najít shodu délky N+1, pokud najde i tu pokusí se najít shodu délky N+2 atd., tedy snaží se nalézt vždy tu nejdelší shodu. Při nalezení shody delší než 0 znaků se do trojice znaků zakóduje znak, který je na pozici N+1.



Obrázek 2.4 Ukázka průběhu algoritmu

a) V zakódované části nejsou žádná data, proto jsme zakódovali data jako trojici 0,0,M.
b) V zakódované části je znak M, avšak znak I nikoliv, zakódujeme trojici 0,0,I.
c) V zakódované části jsou znaky M, I, nikoliv však znak S, zakódujeme trojici 0,0,S.
d) V zakódované části se nacházejí znaky M, I, S tudíž existuje shoda znaku S, avšak žádná delší shoda již neexistuje, zakódujeme trojici 1,1,I.
e) V zakódované části jsou znaky M, I, S, S, I tudíž existuje shoda znaků S, S, I, zakódujeme trojici 3,3,P.
f) V zakódované části jsou znaky I, S, S, I, P, tudíž existuje shoda znaku P, zakódujeme trojici 1, 1, I.
g) Algoritmus se dostal s posuvným okénkem až na konec textu, již není co vyhledávat, proto končí.

Zakódovaná data vypadají následovně:
0,0,M 0,0,I 0,0,S 1,1,I 3,3,P 1,1,I

Obrázek 2.4 však nezobrazuje zcela všechny kroky. Jedná se pouze o jakýsi obraz toho, jak algoritmus funguje. V čem tedy spočívá samostatná komprese? Pokud nalezneme v posuvném okénku shodu délky *N* znaků, jsme schopni ji zakódovat do trojice, odkazující na tuto shodu a tím pádem je šetřeno místo. Čím větší je délka shody, tím více místa je ušetřeno. Ze začátku kódovací sekvence jsou výstupem trojice 0,0,znak a komprese je nulová. Až při postupném naplnění nezakódované části se algoritmus začne odkazovat na již zpracované části textu a kompresní poměr se zvětšuje.

Hledání nejdelší shody v textu je jeden z nejznámějších algoritmických problémů a vyžaduje dost velký výpočetní výkon. Z hlediska efektivity bylo vymyšleno několik různých algoritmů, které zlepšují rychlost hledání nejdelší předpony. Detailní popis fungování algoritmu záleží velice na použitém způsobu vyhledávání redundancí v datech. Je mnoho způsobů, jakým je možné tyto redundance v datech hledat. Některé z přístupů data předzpracují, což většinou následně vede k časovému zrychlení algoritmu. Důležité také je, si rozmyslet, kdy jakou metodu vyhledávání použít. Hledáme-li pouze v krátkém textu, není efektivní data předem zpracovat, protože ve výsledku by mohlo samotné předzpracování brzdit čas potřebný k nalezení shody. Pokud ovšem potřebujeme nalézt shodu v 100000 znacích, je rozhodně moudřejší data předem předpřipravit, protože tato fáze nám ve výsledku ušetří spoustu času.

2.2. Dekódování LZ77

Dekomprese je pak oproti kompresi daleko jednodušší a časově méně náročná. Proto se často LZ77 využívá tam kde je potřeba jednou data zkomprimovat, avšak častěji dekomprimovat. Jediný vstupní parametr, který je třeba k dekompresi znát je velikost kódovacího okénka, v jakém byla data zakódována.

Dekompresní algoritmus postupně čte zakódované trojce a sestavuje dekódovaná data. Nalezne-li dekompresní algoritmus trojici *0,0, znak*, zapíše do souboru pouze dekódovaný znak. Není-li vzdálenost a délka nulová, dekompresní algoritmus se podívá do kódovacího okénka o počet pozic zpět rovný vzdálenosti a odtud zkopíruje znaky. Počet znaků, které kopíruje je roven délce zjištěné z trojice. Jediné co si algoritmus potřebuje udržovat v paměti je pole právě dekódovaných znaků o velikosti kódovacího okénka. Pokud však stroj, na kterém se dekomprese provádí, obsahuje velké množství paměti, není třeba velikost dekódovacího okénka nijak omezovat. Stačí pouze číst data od konce okénka.

K reprezentaci tohoto okénka se využívá cyklický buffer. Funguje podobně jako fronta, tedy nová data se přidávají na konec, ale s tím rozdílem, že je-li počet prvků po přidání větší, než velikost samotného bufferu, je zpředu vymazáno právě tolik prvků, kolik přesahuje velikost bufferu.

Soubor zakódovaný pomocí LZ77 byl vždy dekódován během několika sekund. S mými algoritmy při opravdu velkých souborech byl poměr času komprese vůči dekompresi i více jak 500:1. Časová složitost dekódování LZ77 je tedy skutečně velmi malá.

3. CPU a implementace LZ77

V této kapitole bych lehce poukázal na architekturu samotného procesoru. Následovat pak bude vlastní implementace CPU verze LZ77 včetně různých stupňů optimalizace.

3.1. Architektura CPU

CPU neboli procesor je řídící jednotka každého počítače. Úkolem CPU je zpracovávat posloupnost instrukcí, podle kterých vykonává určité operace. Procesor spolupracuje s pamětí, z které načítá nebo ukládá právě zpracovávaná data. Data jsou předávány z procesoru do paměti pomocí paměťové sběrnice. Procesor je velmi složitý integrovaný obvod, obsahující řádově miliony tranzistorů. Procesor se jeví jako prvek v počítači schopný sekvenčně zpracovávat data.

Do roku 2005 byly pro veřejnost dostupné pouze procesory s jedním fyzickým jádrem. V praxi to znamená, že procesor je schopen v jedné chvíli zpracovat pouze jednu instrukci. Roku 2006 byly na trh uvedeny první více jádrové procesory. Výhodou těchto procesorů je, že každé jádro je schopno během jednoho okamžiku zpracovávat své vlastní instrukce, což vede k možnosti paralelizace. Je jasné, že díky více fyzickým jádrům v procesorech, se začalo přemýšlet o paralelizaci algoritmů.

Procesor se jeví jako jeden monolitický čip s velkým výpočetním výkonem. Tomu bude také odpovídat samostatná implementace algoritmu LZ77. Data budou zpracovávána sekvenčně od začátku až do konce. Díky běžnému používání více jádrových procesorů u stolních i přenosných počítačů jsem zahrnul i více jádrovou implementaci pro CPU, která s sebou ale nese jistá omezení. Nicméně při dodržení jistých podmínek ji lze pěkně využít ke zrychlení celého kódování.

3.2. Vlastní implementace LZ77

3.2.1. Pseudokód LZ77

```
pro celou délku vstupního souboru
{
      vytvoř vyhledávací okénko S velikosti M
      pro maximální délku nezakódované části okénka
      {
            vytvoř nezakódované okénko P velikosti M
            pokud nenalezneš žádný výskyt prvního znaku P v okénku S
                  zapiš na výstup 0,0, a nenalezený znak
                  zastav vnitřní cyklus a posuň se o jeden znak dále
            pokud existuje první znak P v S
                  ulož nejdelší shodnou sekvenci
                  pokus se najít delší sekvenci znaků
            jinak
                  zapiš vzdálenost od aktuálního indexu ke shodě
                  zapiš délku shody
                  zapiš písmeno následující v S za shodou
                  posuň se v textu o počet znaků rovnající se délce
                  shody
      }
}
```

V pseudokódu je šedou barvou naznačena část, která je na vlastním algoritmu nejsložitější. Díky funkci performance wizard ve Visual studiu 2010 jsem byl schopen analyzovat čas jednotlivých operací. Nalezení vlastní shody mezi *P* a *S* zabere více jak 95% času celého algoritmu. V definicích optimalizací bude popsáno několik způsobů, jak shody vyhledávat efektivněji.

Teď už víme, jak algoritmus funguje, takže můžeme přikročit k samostatné implementaci. K naprogramování CPU verze LZ77 jsem využil jazyk C# a vývojářský nástroj Visual studio 2010. Algoritmus by měl být navržen tak, aby byl schopen kódovat text o velikosti několika desítek MB s posuvným okénkem o velikosti až 4MB.

Při spuštění programu je nutné jako vstupní parametr zadat cestu k souboru a velikost kódovacího okénka. Data jsou načtena do paměti RAM, a jsou reprezentována jako pole bajtů. V dřívějších implementacích jsem soubor načítal jako řetězec, avšak zde nastal problém se zalamováním řádků, proto bylo nutné soubor načítat jako pole bajtů. Po načtení souboru z pevného disku se vytvoří v paměti vyhledávací okénko *S*, s přednastavenou velikostí *N*. V tomto okénku se uchovávají data, v kterých budeme hledat shodu.

K vytvoření ohraničení tohoto okénka jsem využil iterační proměnou cyklu, díky které jsem schopen nastavit jak pozici okénka v textu, tak i jeho délku. Důležité je, že data nejsou předávána vyhledávacímu algoritmu jako celek, nýbrž je vytvořeno nové pole, do kterého je zkopírována podmnožina dat od určité pozice s určitou délkou. V cyklu, který se stará o procházení všech znaků textového souboru je zanořen další cyklus, který má za úkol nachystat data, která budeme v posuvném okénku hledat. Opět se jedná o pole, do kterého jsou překopírovány požadované znaky.

Obrázek 3.1 naznačuje, jak vypadá samostatná implementace procesu hledání. Modrá barva značí vyhledávací okénko, žlutá čára zobrazuje aktuální iterační proměnou vnějšího cyklu, tedy předěl mezi zakódovanou a nezakódovanou částí okénka. Buňky zelené barvy reprezentují zatím nejdelší nalezenou shodu. Je-li nalezen znak „K“ v zakódované části okénka, algoritmus zvětší hledaný podřetězec na „KA“ a pokusí se jej nalézt. Pokud se mu to podaří, hledaný podřetězce se opět zvětší na „KRÁ“. Tento proces se opakuje do té doby, dokud v nezakódované části existuje celý podřetězec. Jak lze vidět na řádku číslo 5, existuje neshoda mezi znakem „_“ a „O“, tedy *S[4] != P[4]*, z čehož vyplývá, že ve vyhledávacím okénku neexistuje podřetězec „KRÁLO“. Do výstupu tedy kódujeme trojici *7,4,O*.

K	R	Á	L		A		K	R	Á	L	O	V	N	A
K	R	Á	L		A		K	R	Á	L	O	V	N	A
K	R	Á	L		A		K	R	Á	L	O	V	N	A
K	R	Á	L		A		K	R	Á	L	O	V	N	A
K	R	Á	L		A		K	R	Á	L	O	V	N	A
K	R	Á	L		A		K	R	Á	L	O	V	N	A

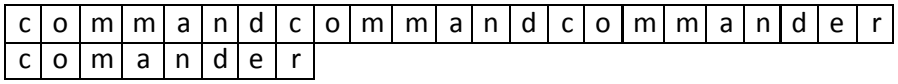
Obrázek 3.1 Hledaní nejdelšího řetězce

3.3. Definice optimalizací vyhledávání

Rychlost algoritmu samotného záleží na několika faktorech. První a nejvýznamnější faktor, který ovlivňuje rychlost kódování je velikost vyhledávacího okénka. Dalším faktorem ovlivňujícím rychlost je samotná velikost vstupního souboru. Mějme vyhledávací okénko S o velikosti M znaků. Nechť hledaný řetězec P je délky N a první shodný znak P a S se nachází v S na pozici C.

3.3.1. Naivní algoritmus

Pro nalezení shody můžeme zkusit následující způsob. Začneme porovnáním prvního znaku z S s prvním znakem z P. Pokud se znaky neshodují, posuneme se o jeden znak v S dále a opět porovnáme. Takto opakujeme, dokud nenalezneme shodu. Pokud nalezneme shodu, tedy $S[C] = P[0]$, pak zkusíme porovnat znaky $S[C+1]$ a $P[1]$. Pokud se tyto znaky shodují, posuneme se o jeden znak v S i P a ten dále porovnáme. Při shodě proces opakujeme tolikrát, jaká je délka hledaného řetězce. Při neshodě znaků se posuneme o jednu pozici v S dále a opakujeme porovnávání. Hledaný řetězec P je nalezen, pokud počet za sebou shodných znaků je N. Tento způsob nazývaný taktéž naivní algoritmus je avšak velmi neefektivní. Časová složitost v nejhorším případě je tudíž $O((N - M + 1) * M)$.



Obrázek 3.2 Ukázka nejhorší možné situace pro hledání vzorku

Výhody

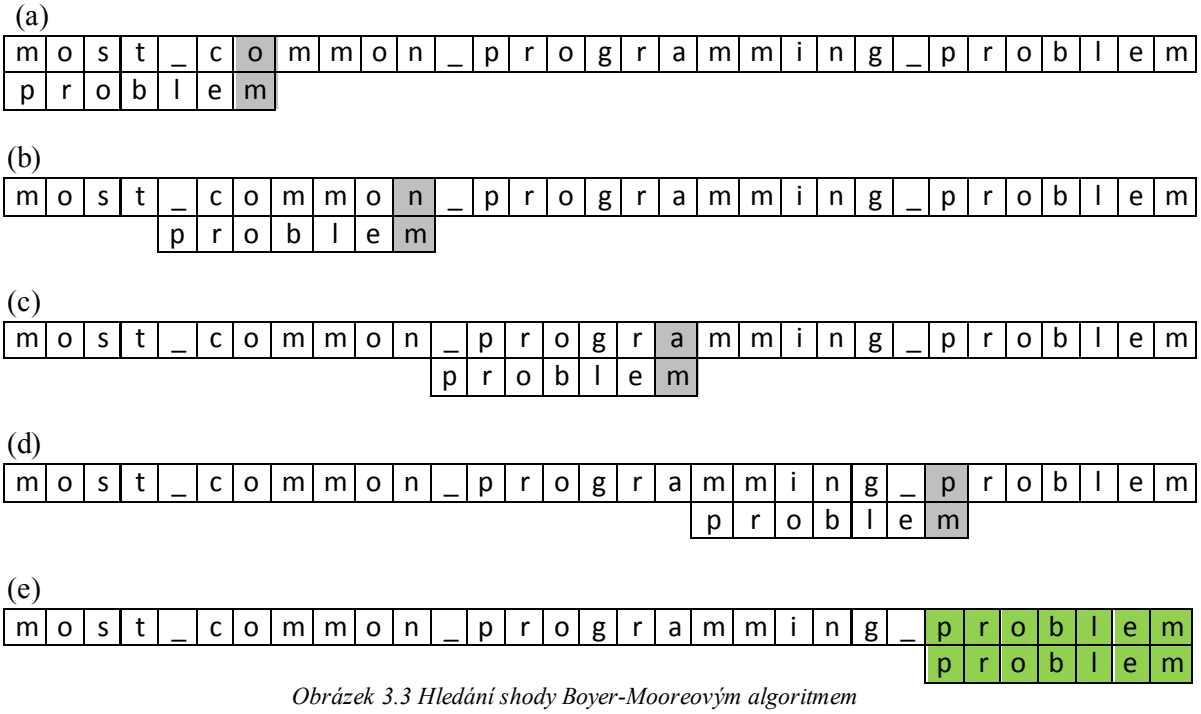
- Jednoduchá implementace
- Užitečný pouze pro vyhledávání v krátkých textech
- Není třeba data předzpracovat

Nevýhody

- Dlouhá doba nalezení shody
- Nepočítá s informacemi získanými z předchozích kroků
- Nevhodný pro velké slovníky

3.3.2. Boyer-Mooreův algoritmus

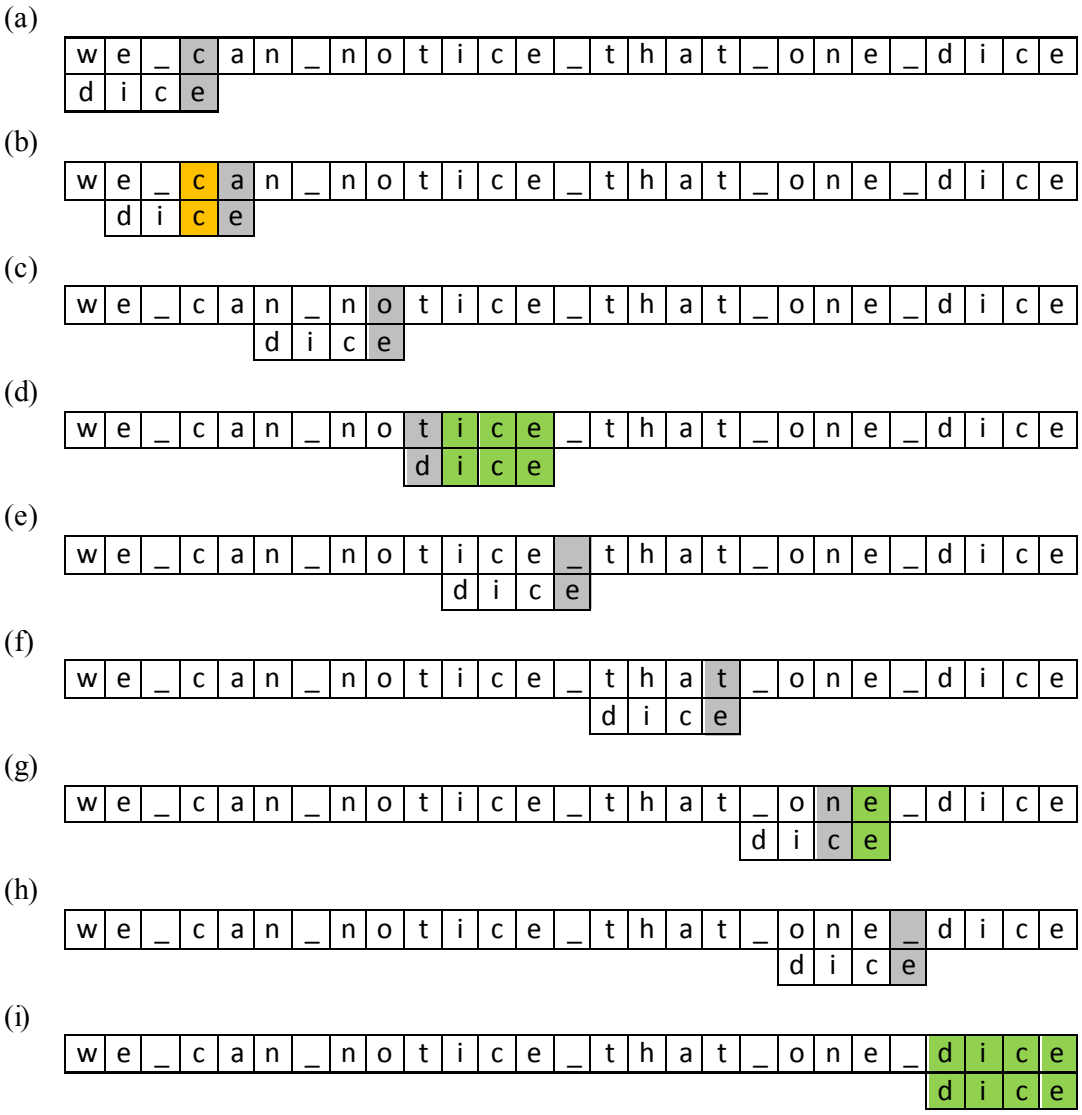
Jelikož je naivní algoritmus časově velice náročný a neefektivní byl vymyšleno několik dalších algoritmů jak text efektivněji prohledávat. Jedním z nich je právě tento algoritmus, který vymyslel S. Boyer a J. Strother Moore [2]. Tito pánové zjistili, že není potřeba srovnávat každé písmeno z vzorku s prohledávaným textem, nýbrž stačí porovnat jen znaky, které jsou jistým způsobem důležité. Jelikož budeme porovnávat znaky důležité, je nutné si tyto důležité znaky předem nějak určit. Proto se před samotným začátkem algoritmu musí předzpracovat tabulka skoků špatných znaků. Z anglického „bad character shift table“. Představme si úryvek text „most common programming problem“, v kterém potřebujeme nalézt řetězec „problem“.



Obrázek 3.3 Hledání shody Boyer-Mooreovým algoritmem

Na začátku algoritmu je hledaný vzor zarovnán zepředu s textem. Algoritmus opět prohledává text zleva doprava, avšak vzor je porovnáván s textem odzadu, tzn. zprava doleva.

Důležité písmeno v našem vzoru je poslední písmeno, tedy písmeno „m“. V první fázi (a) začnu tedy porovnávat znak „m“ se znakem „o“. Jelikož se znaky nerovnají, posunu vzorek v texu o N pozic doprava. Počet znaků o kolik se mohu v textu posunout je uložen v tabulce skoků. Ve fázi (b) opět porovnám poslední písmena. Znak „m“ se neshoduje se znakem „n“, tudíž musím vzorek opět posunout. Stejně tak, je posun naznačen i ve fázi (c) a (d). Ve fázi (e), existuje shoda mezi „m“ a „m“. Při shodě posledního znaku se algoritmus posune na další písmeno, tedy doleva, a to porovná. Shoduje-li se každý další znak, proces se opakuje. Není-li nalezena žádná neshoda v textu a vzorku, je vzorek nalezen a algoritmus končí. Zde je vidět jedna z extrémních situací, které mohou nastat a to, že není třeba porovnávat znaky všechny, nýbrž jen poslední znak. Jedná se o ideální případ, protože pro nalezení jsme potřebovali provést pouze 11 porovnání oproti 31 porovnáním, které by musel provést naivní algoritmus. V praxi se však tento případ vyskytuje vcelku zřídka. Vždy se však nepodaří text přeskakovat o N znaků vpřed a tím pádem urychlit vyhledávání. Situace, která nastává velmi často, je zobrazena na obrázku.



Obrázek 3.4 Hledání shody B-M algoritmem s využitím posunu špatného znaku

Nyní chceme v textu najít vzor „dice“. Opět zarovnáme vzorek na začátek textu. Porovnáme poslední písmeno, a při nalezení neshody, se posuneme o jeden znak vpřed. Důvod, proč posunujeme vzorek jen o jeden znak dopředu je zřejmý. Ve fázi (b) se totiž v textu nachází písmeno „c“. Proto existuje možnost, že pokud zarovnáme text i vzorek vzhledem k písmenu „c“ můžeme nalézt shodu. Jak je ale vidět, poslední písmeno vzorku se neshoduje s písmenem v textu a proto se posunujeme o N znaků vpřed. Ve fázi (d) byla nalezena shoda délky N-1, avšak první písmeno vzorku se neshoduje s textem, provádíme další posun o 1 znak vpřed, díky možnosti výskytu vzorku v textu. Při neshodnosti posledních znaků se opět posunujeme o N pozic vpřed. Ve fázi (g) existuje shoda posledních znaků, avšak znaky předposlední se neshodují, algoritmus se tedy posune o jeden znak vpřed, abychom měli naprostou jistotu, že nebyl vynechán žádný důležitý znak. Ve fázi (i) byly porovnány všechny znaky a byla nalezena shoda. Shodu se podařilo nalézt po 16 porovnáních, oproti naivnímu algoritmu, který by provedl 27 porovnání.

Je jasné, že nejlepší rychlosti prohledávání textu je dosaženo, čím delší je hledaný vzor a výskyt písmen stejných, jako je ve vzorku je velmi malý, či absolutně žádný. Při ideálních situacích je totiž přeskočeno právě N znaků, což značně snižuje čas, pro nalezení shody. Zato při hledání vzorku o délce jednoho či dvou znaků není ušetřen skoro žádný čas. I přes tyto nevýhody je však v praxi tento algoritmus 5-8x rychlejší než klasické naivní prohledávání textu. Existuje celá škála algoritmů pro vyhledávání v textu, ale právě Boyer-Mooreův algoritmus je považován za jeden z nejrychlejších vyhledávacích algoritmů. Díky tomu je velice často implementován v různých pokročilých textových editorech. Časová složitost je v nejhorším případě $O(n + m)$, za předpokladu, že se vzorek v textu nenachází.

Výhody
- Rychlost oproti naivnímu hledání je 5x až 8x větší
- Relativně jednoduchý na implementaci

Nevýhody
- Malá rychlost nad malou abecedou

3.3.3. Využití informací z minulých hledání

Mějme zakódovanou část textu *S*, o velikosti $P = 4$ milióny znaků. Představme si následující situaci. Kodér právě zakódoval trojici 0,0,znak, která značí, že se právě zpracovaný znak nikde v zakódované části nenachází. Nyní chceme nalézt shodu délky N, proto musíme začít zakódovanou část prohledávat od začátku zakódované části. Nechť se shodný znak nachází na pozici M. Po nalezení shody délky N se vyhledávání spustí znovu, ale nyní s délkou hledaného řetězce N+1. Při naivním zpracování začneme text opět prohledávat od začátku, a nebudeme vycházet z informace získané v předchozím kroku, a to že, první shodný znak se nachází na pozici M. Díky tomu, že algoritmus najde vždy první výskyt hledaného znaku či řetězce, jsme s jistotou schopní říci, že nemusíme v další iteraci hledat nový řetězec zcela od začátku, ale právě od pozice M. Rychlost zpracování algoritmu se radikálně zvýší, použijeme-li takto získané informace. Pro názornost jen malé srovnání. Budeme-li v *S* hledat shodu *R* délky 10 pomocí naivního vyhledávání a výskyt prvního znaku se bude nacházet na pozici P-10.

- Při ignorování získaných informací z minulých hledání

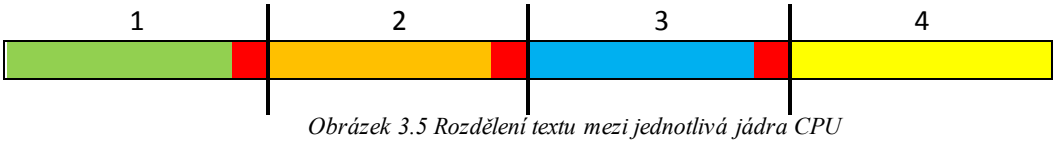
Budeme muset provést 10 prohledávání nad P-10 znaky. Pro nalezení shody je tedy potřeba provést: $10 * (P - 10) = 3{,}99 * 10^7$ porovnání.

- Při využití informací získaných z minulých hledání

Stačí udělat $(P - 10) + 9 = 3{,}99 * 10^6$ porovnání. To znamená o jeden řád porovnání méně, neboli nám k nalezení shody stačí pouhá jedna desetina počtu porovnání. Tuto informaci je tedy rozhodně dobré použít.

3.3.4. Paralelní CPU verze LZ77

Ačkoliv smyslem této práce nebylo vymyslet paralelní verzi LZ77 pro CPU, tak díky jednoduché úpravě zdrojového kódu bylo možné spustit kódování paralelně. Paralelismus zde funguje na zcela základním principu, kdy je vstupní soubor rozdělen na části dle počtu jader a následně je každým jádrem zpracovávána část textu. Každé jádro kromě prvního musí vyhledávat shody i v předchozím bloku textu a to do vzdálenosti velikosti nezakódované části. Představme si okénko o velikosti 64kB. Kvůli zachování stejného kompresního poměru jako u jedno jádrové verze, bylo nutné zajistit, aby každé další jádro kromě prvního bylo schopno vyhledávat data i v textu patřící jádru předchozímu. V tomto případě tedy jádro 2 vyhledávalo shody i v posledních 64kB textu patřících předchozímu jádru. Stejně tak to platí pro jádro 3 a 4. Kdybychom nedodrželi překrývání textu, tak by každá výstupní sekvence daného jádra začínala trojicemi 0,0,znak. Maximální komprese v rámci jedné části by tedy bylo dosaženo až po znaku 64K. Obrázek 3.5 ukazuje, rozdělení textu mezi 4 jádra CPU.



Obrázek 3.5 Rozdělení textu mezi jednotlivá jádra CPU

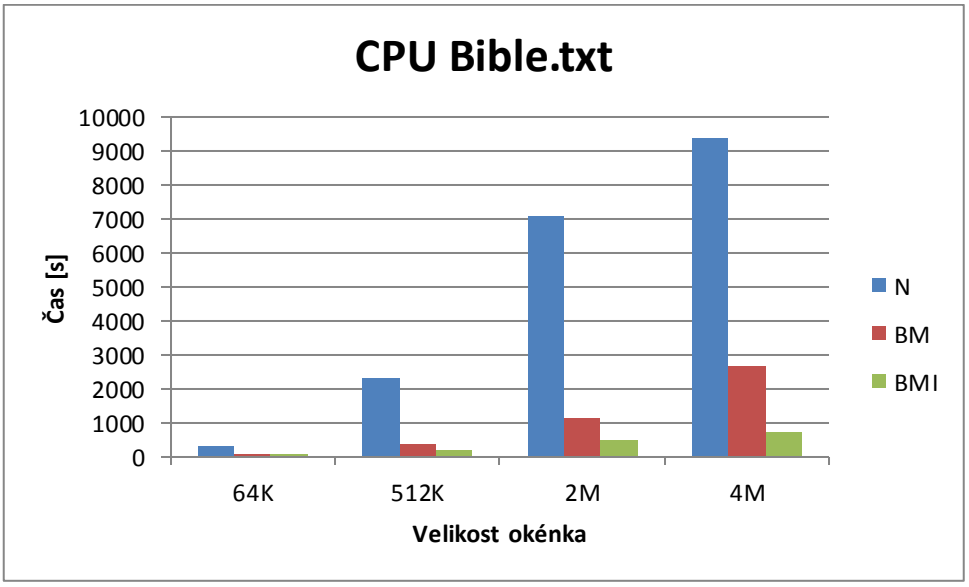
3.3.5. Srovnání CPU verzí LZ77

CPU verze LZ77 jsem testoval na procesoru Intel Core i7 2670QM a starším procesoru Intel Core 2 Duo E8400. Frekvence procesorů byla u 2670QM 3.0GHz, u E8400 3,4GHz. Jedno jádrová verze byla testována pouze na procesoru E8400.

Získané výsledky byly naměřeny na těchto verzích:

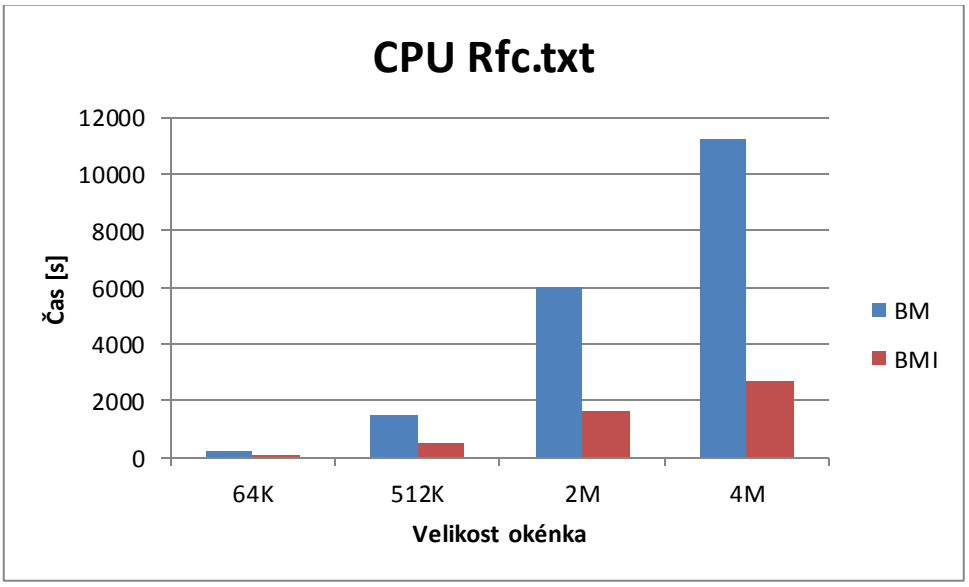
- Naivní (N)
- Boyer-Mooreova (BM)
- Boyer-Mooreova s využitím informace z předchozího hledání (BMI)
- Paralelní CPU verze B-M s využitím informace (BMI)

Testovaným textem pro celou tuto práci byla anglická bible, 4MB, část z textu RFC (protokoly internetové komunikace), 10MB, a část textu LAW, 20MB, právního spisu České Republiky. Velikost vyhledávacího okénka byla 64kB, 512kB, 2MB, 4MB.



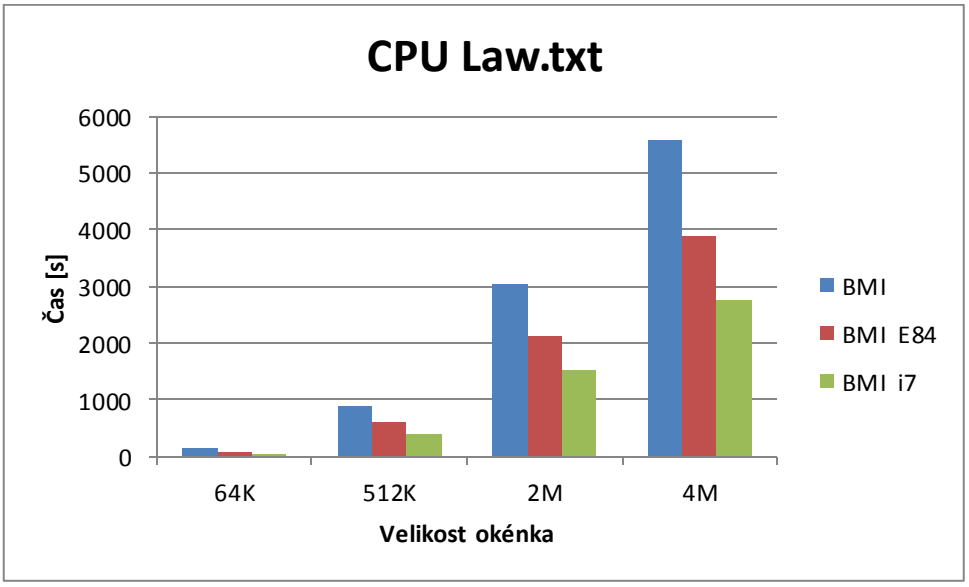
Obrázek 3.6 Rychlost CPU algoritmů nad souborem bible.txt

Naivní verze je časově velmi neefektivní a při vyhledávacím okénku o velikosti 4MB je čas nutný ke zpracování více jak 150 minut. Cestou naivního vyhledávání tedy nemá smysl větší soubory kódovat. BM verze je až 6x rychlejší oproti verzi N. Doba komprese nejefektivnější verze BMI je cca. 10 minut. BMI je tedy oproti naivnímu vyhledávání více jak 15x rychlejší, oproti BM cca. 3-4x. Pro větší soubory je rozdíl ještě markantnější.



Obrázek 3.7 Rychlost CPU algoritmů nad souborem rfc.txt

Časová složitost BM verze je opět 3x větší, než je tomu v případě verze BMI. V posledním případě, kdy velikost okénka je 4M znaků je rychlost BMI již 4x větší. Opět zde platí pravidlo, že pro větší soubory je rozdíl markantnější. Jelikož je verze BM také neefektivní, nebude již dále zahrnuta v grafech. Pro lepší přehlednost bude verze BMI srovnána se svou paralelní CPU verzí.



Obrázek 3.8 Rychlost CPU algoritmů nad souborem law.txt

Do testování paralelní CPU verze jsem zahrnul dvou jádrový Core 2 Duo E8400 a osmi jádrový Core i7 2670QM (4 fyzická jádra). Jedno jádrová verze BMI je testována na procesoru E8400.Rychlost komprimace paralelního algoritmu s Core i7 2670QM je tedy 2x větší. U paralelní verze pro Core 2 Duo E8400 je rychlost 1.5x větší.

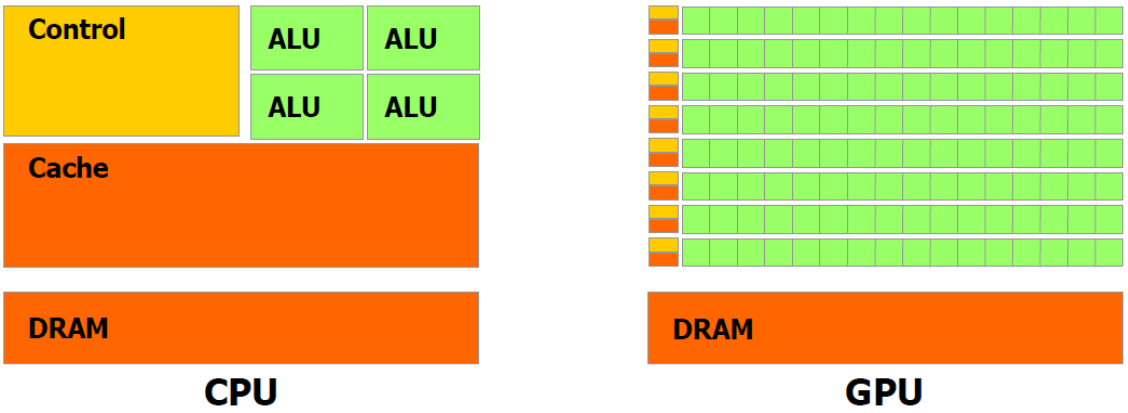
4. GPU a implementace

Pro dobré pochopení GPU verze algoritmu je nutné nejdříve alespoň trochu vysvětlit architekturu samotného GPU. Samotná GPU implementace totiž velmi úzce souvisí s danou architekturou. Úkolem je tedy prozkoumat možnosti GPU v oblasti vyhledávání shod a následné komprese, včetně optimalizací.

4.1. Architektura GPU

GPU neboli grafická karta je zařízení, starající se o výstup dat z počítače na monitor. V dřívějších dobách byla grafická karta určena pouze k této činnosti. Později s rozmachem počítačové techniky a náročnosti aplikací se z grafické karty stal akcelerátor, který pouze nepřeposílal data z počítače na monitor, ale také sám zpracovával 2D a 3D scénu. V dnešní době jsou grafické karty masivně paralelní čipy, které se využívají v různých vědeckých odvětvích jako je matematika, biologie či simulace přírodních jevů. Dnešní GPU jsou navrženy tak, aby bylo možno zpracovávat daný úkol současně na více procesorech. Tomuto se říká paralelismus. Lze si to představit jako zpracování neustále stejných operací nad velkým množstvím dat. Zde je dobré si ukázat, jak taková paralelizace funguje.

Představme si dvě velká pole v paměti o velikosti deseti miliónů prvků. Procesor bude muset sekvenčně projít přes všechny prvky pole a postupně je posčítat. To může trvat nějakou dobu. Grafická karta tuto úlohu řeší jinak. Díky stovkám jader je schopno toto velké pole rozdělit, a každému jádru přidělit právě jeho část, kterou jádro spočítá. Díky tomu, že každá jednotka pracuje jen s malým množstvím dat, je čas pro spočítání výrazně nižší než u klasického CPU. Architektura GPU je zcela jiná, než architektura CPU. Obrázek 4.1 ukazuje, že GPU neobsahuje tak velkou kešovanou paměť jako CPU, více se zaměřuje na velký počet relativně jednodušších jednotek, než na jeden monolitický čip o velkém výkonu.



Obrázek 4.1 Porovnání architektury CPU a GPU [3]

4.2. nVidia CUDA

Technologii CUDA si můžeme představit jako programovací API pro grafickou kartu. Jelikož je vyvinuta firmou nVidia, není možné ji spouštět na kartách ostatních konkurenčních firem. Primárně pro složité výpočty je určena řada grafických karet nVidia Quadro, či řada karet nVidia Tesla. Od grafických karet řady 8 je již možno tyto výpočty provádět i na běžných

řadách karet typu GeForce. CUDA je platformě nezávislá, což znamená, že lze vyvíjet aplikace pro grafické karty pod jakýmkoli operačním systémem jako je MAC OS, Linux či Windows. K naprogramování může být použit jazyk jako je Fortran, OpenCL, Direct Compute či často používané C s minimálním rozšířením. Důvodem použití jazyka C je častá znalost programátorů, tudíž se programátoři mohou soustředit na programování, a nemusí se učit novému jazyku.

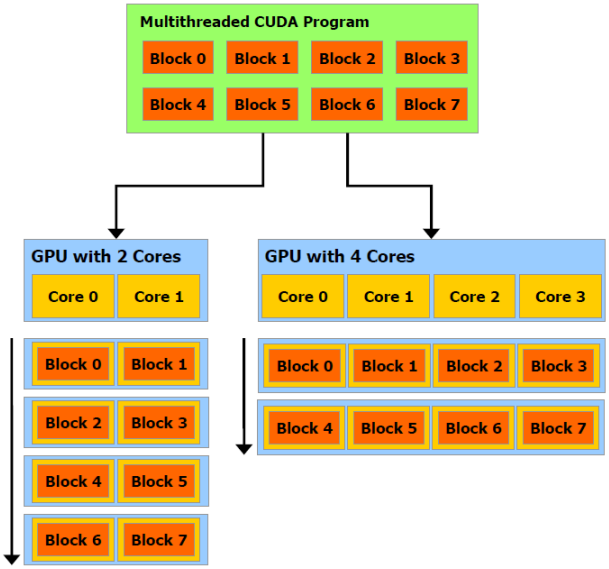
Samostatný program je rozdělený do bloků vláken, které jsou na sobě nezávislé, takže GPU s více jádry je schopno spustit program na více blocích najednou a provést výpočet rychleji. Samostatná vykonávací funkce zvaná v GPU kernel je tedy realizován několika paralelními vlákny (threads). Programátor uspořádá tato vlákna do bloků vláken (blocks) a ty pak do vláknové mřížky (grid). GPU si pak rozloží jádro programu do mřížky paralelních bloků. Každé vlákno v bloku bude spuštěno tolikrát, kolik nastaví programátor vláken v bloku. Každé vlákno obdrží svůj unikátní identifikátor (ID). Celý blok vláken může obsahovat až 1536 samostatných vláken, a celému bloku je také přidělen identifikátor (ID), ze kterého je gridu. Celý tento grid je vykonáván jediným jádrem programu a výsledek může být zapsán a čten z hlavní paměti. Pro jednoduché určení identifikátorů připravila nVidia v každém kernelu tyto vlastnosti.

- threadIdx jednoznačný identifikátor vlákna
- blockIdx identifikátor bloku
- gridDim identifikátor mřížky

Všechny tyto vlastnosti mohou být typu dim3 (x,y,z), což je užitečné při práci s 3D polem. Pokud programátor nepotřebuje všechny tři rozměry, stačí pouze specifikovat rozměr dimenze x nebo y a překladač implicitně dosadí za nevyplněnou hodnotu rozměr velikosti 1. V samotném kernelu se pak přistupuje ke složkám x,y,z za pomoci tečkové notace.

4.3. Programování GPU

Grafická karta je schopna fyzicky spustit několik stovek, virtuálně však až několik tisíc vláken najednou [4]. S tímto souvisí jistá omezení. Vlákna musí být na sobě nezávislá, protože není známo, v jakém pořadí se budou vykonávat. Grafická karta je tedy více určena pro intenzivní výpočty s malým počtem podmínek. Aby bylo dosaženo dobré škálovatelnosti grafických čipů, bylo nutné vymyslet logické celky, díky kterým lze kód efektivně rozprostřít mezi jednotlivá vlákna a výpočet tak urychlit.



Obrázek 4.2 Automatická škálovatelnost [3]

Samostatný kód se dělí na dvě části. První část je klasický sériový kód vykonávaný na CPU. Ten se označuje termínem host. Druhou část tvoří paralelní kód, spouštěný na grafické kartě. Označujeme termínem device. Tyto oba celky můžeme libovolně prolínat, kompilátor při překladu sám pozná, které části patří procesoru a které grafické kartě. Grafická karta má několik druhů pamětí [3].

Globální paměť, v dnešní době o velikosti stovek MB, vyšší edice GPU obsahují i několik GB globální paměti. Tato paměť je velká, avšak přístupová doba je vcelku vysoká cca. 600 strojových cyklů. Dále se tato paměť člení na **lokální paměť**. Ta se začne využívat při situacích, kdy dojde k tomu, že námi napsané kernely přesahují množství paměti pro ně určené. Jelikož je ale tato paměť pomalá, měli bychom si dát pozor a takovýmto situacím se vyvarovat. Paměť pro **konstanty**, která využívá malou cahce paměť, do které načte hodnoty z globální paměti. Tuto cache pak zpřístupní všem vláknům, což může být výhodné tam, kde je potřeba načítat stejná data pro více vláken. **Sdílená** paměť je společná pro všechny vlákna v jednom bloku, avšak její kapacita je značně omezená. Hlavním výhodou je velice nízká přístupová doba, udává se pár jednotek strojových cyklů a je vhodná pro souběžný přístup. Posledním druhem paměti jsou takzvané **registry.** Jedná se paměťový prostor vymezený pro každé vlákno spuštěné v jednom bloku. Rychlost paměťových registrů je také velmi vysoká, zhruba jako u sdílené paměti. Bohužel je počet registrů pro každý kernel velice malý. Pokud nastane situace, že jsou zaplněny všechny registry pro daný kernel, začne být automaticky používána paměť lokální, což vede k drastickému snížení výkonu celé GPU.

K vytvoření vlastního kernelu nám slouží identifikátor __global__ s návratovým typem void. Není totiž možné z kernelu vracet výsledky pomocí návratové hodnoty, nýbrž se výsledky musí překopírovat z paměti device do paměti hosta. Průběh výpočtu se pak odehrává podle následujícího scénáře:

- Inicializace dat na hostovi
- Alokace paměti na GPU
- Překopírování dat z paměti do globální paměti GPU
- Spouštění paralelního výpočtu
- Překopírování dat z paměti GPU do paměti hosta
- Uvolnění zdrojů

4.4. GPU verze LZ77

Jelikož je architektura grafické karty zcela odlišná od klasického procesoru, bylo nutné vymyslet jiný způsob zpracování dat. Jak už bylo řečeno, nejslabší místo celého algoritmu je samotné vyhledávání shod. Abychom byli schopni algoritmus urychlit, musíme najít způsob jak efektivně hledat shody paralelně. Aby byla práce s daty rychlá, je nutné nakopírovat data jako celek do paměti GPU na začátku a s nimi dále pracovat. Obecně vzato je vždy lepší nakopírovat jeden velký blok dat najednou, než kopírovat vícekrát malé bloky dat. Abychom byli schopni naprogramovat efektivní algoritmus, je třeba mít dost velké znalosti o architektuře grafické karty. Pro testování všech algoritmů na GPU jsem využil 256 vláken v rámci bloku. Důvodem byl nejvyrovnanější výkon.

4.4.1. GPU naivní verze LZ77

V rané fázi jsem se zaměřil pouze na tu část algoritmu, kde se vyhledávají shody. To znamená, že celý postup s připravením posuvných okének byl stejný, jako na procesoru. Tedy dva v sobě vnořené cykly, z nichž vnější cyklus vytvářel indexy, které ukazovaly na začátek a konec prohledávacího okénka. Vnitřní cyklus vytváří indexy, které odkazují na to, co se bude vyhledávat. Indexy se následně předají kernelu na GPU, včetně ukazatele na prohledávaná data. Dle aktuální délky velikosti prohledávacího okénka byl vždy spuštěn potřebný počet bloků k prohledání. Pokud byla tedy velikost okénka 64kB, bylo spuštěno 256 bloků s 256 vlákny v každém bloku. Každé vlákno v rámci bloku se zarovnalo vzhledem k textu tak, aby bylo možné při případné shodě jednoznačně identifikovat vzdálenost od nezakódované části. Obrázek 4.5 zobrazuje, jak rozmístění bloků nad textem vypadá.

Při nalezení shody mezi vláknem a prvním znakem nezakódované části se vlákno v textu posunulo o jeden znak vpravo, stejně tak i v části zakódované. Při první neshodě znaků byla do výsledného pole zapsána délka shody. V případě kdy nebyl shodný ani jeden znak byla zapsána hodnota -1, která značí neshodu znaků. Následně byly výsledky zpracovány pomocí paralelní redukce, a to přímo z knihovny thrust z SDK od firmy nVidia. Ta přebírá jako vstupní parametr odkaz na začátek prohledávaného pole, konec prohledávaného pole a enumerátor pro zvolení operace, kterou chceme nad daty provést. Detailní postup paralelní redukce bude popsán níže v této práci, včetně vlastní implementace. Výstupem redukce již byla hodnota, značící index začátku shody. Pokud tato hodnota nebyla -1, vyhledávání se opakovalo, avšak s velikostí nezakódované části o jeden znak větší. Tento algoritmus se však časem ukázal jako velmi

neefektivní, ale právě díky jeho slabým stránkám jsem byl schopen další implementace vylepšovat.

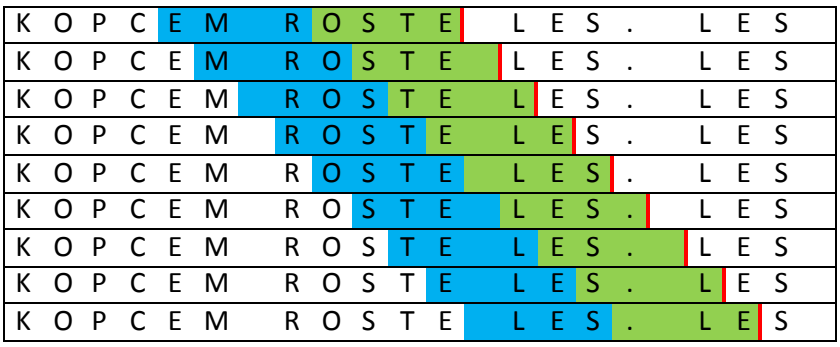
4.4.2. Masově paralelní GPU verze LZ77

Největším zpomalením první verze bylo opakované hledání stále stejných shod a využívání pouze globální paměti. Při nalezení shody délky *N* se algoritmus pokusil nalézt shodu délky *N+1*, což vedlo k opětovnému vyhledávání již nalezených shod, jen s tím rozdílem, že se hledala data o jeden znak delší. Z hlediska paralelizace to vůbec nebyl správný krok. Proto jsem zkusil následující postup. Posuvné okénko, jak už víme, se dělí na dvě části. Levá již zakódovaná část a pravá ještě nezakódovaná část. Mezi nimi je index, který značí aktuální polohu okénka v textu. Tento pomyslný index byl tedy umístěn paralelně na všechny možné pozice v textu. Každý index rozmístěný v textu byl v GPU reprezentován jedním blokem. Tedy nad textem o délce jednoho milionu znaků, bylo paralelně vytvořeno jeden milion posuvných okének. Jelikož je maximální velikost bloku na mé grafické kartě v jednom rozměru 65000, bylo nutné nastavit i druhý rozměr.

Zde již nastal první problém. Každý text má jinou velikost a musel být nalezen vždy společný násobek dvou čísel, který se rovná velikosti soboru. Avšak ani jeden z operandů nesměl být větší než 65000. Pokud by byla velikost souboru prvočíslo, nastal by problém.

Každý blok obsahoval 256 vláken, kde každé vlákno reprezentovalo jeden znak v textu. Jelikož je počet maximálně spouštěných vláken v bloku dán pevně architekturou, nebylo tedy možné vyhledávat redundance v textu dále než 256 znaků od aktuálního indexu. Proto bylo nutné kernel upravit, a vytvořit cyklus, který toto zařídil. Byla-li velikost okénka 2048 znaků, byl cyklus spuštěn $2048/256 = 8$ krát.

Obrázek 4.3 zobrazuje rozložení bloků nad textem. Pro jednoduchost obsahuje každý blok 4 vlákna, a velikost vyhledávacího okénka je 8 znaků. Cyklus je tedy spuštěn 2x pro každý blok.



Obrázek 4.3 Paralelní rozložení bloků nad textem

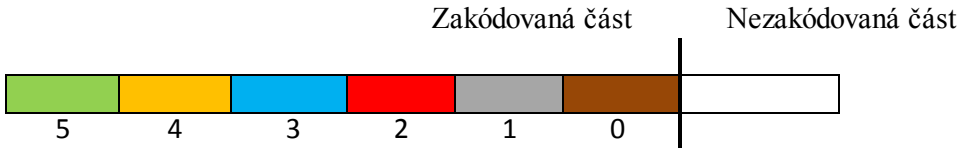
Algoritmus fungoval tak, že byla vždy nalezena nejdelší shoda v rámci počtu aktivních vláken v bloku. Ta byla uložena a cyklus se spustil znovu pro další znaky. Následně byly porovnány maximální shody v rámci bloku. Po dokončení prohledávací fáze bylo spuštěno prohledání výsledků. Výsledky z grafické karty byly překopírovány do paměti hosta. Následně procesor sekvenčně prohledával výsledné pole. Narazil-li na trojici s délkou maximální shody 0, pouze zapsal do výstupního souboru trojici 0,0,znak. Pokud byla délka větší jak 0, byla zapsána

shoda ve tvaru offset, délka, následující znak a celé prohledávání se posunulo právě o délku nalezené shody. Tento způsob byl avšak neefektivní.

Představme si následující situaci. Nechť je délka textu *N* znaků. Počet spuštěných bloků je tedy také *N*. Nechť *M* je nějaký blok spuštěný v textu, kde délka nalezené shody není nula. Byla-li algoritmem nalezena nějaká shoda délky *K*, pro blok *M*, pak všechny další shody *M+K* jsou neplatné, protože nejsou do výstupního souboru kódovány. Prakticky tedy nalezne-li blok s indexem 500 shodu délky 10, pak všechny další bloky s indexem 501-510 byly spuštěny zbytečně, protože jejich výsledky nebudou ve výstupním souboru zahrnuty. Je tedy jasné, že tento algoritmus je značně neefektivní. Lze jednoduše spočítat, že pokud délka shod v celém souboru se rovná *L*, pak bylo spuštěno zbytečně *L* bloků prohledávání. Pro malé soubory s malými okénky byl tento algoritmus rychlejší než jeho předchůdce. Bohužel při větších soborech se stal mnohonásobně pomalejším než předchůdce.

4.4.3. GPU optimalizovaná verze LZ77

Díky neefektivnosti předchozích algoritmů jsem byl nucen změnit způsob vyhledávání shod v datech. Bylo nutné opravit nedostatky předchozí verze. V části kódu, který řídí procesor, se pouze spočítal aktuální index, od kterého bude grafická karta vyhledávat. Vyhledávání však probíhá zcela jinak. Představme si následující situaci. Nechť velikost vyhledávacího okénka je 64kB. Abychom byli schopni vyhledávat ve všech 64kB textu, je nutné spustit 64536/256 bloků. Každému bloku bude přidělena právě 1/256 celého vyhledávacího okénka. Blok s indexem 0 tedy bude vyhledávat v okénku od indexu 255/256 až po index 256/256. Blok 1 od indexu 254/256 – 255/256, atd. Obrázek 4.4 zobrazuje, jak vypadají jednotlivé bloky spuštěné nad textem. Bílá barva reprezentuje text, který chceme hledat, ostatní barvy zobrazují jednotlivé částí vyhledávacího okénka rozdělené mezi bloky.



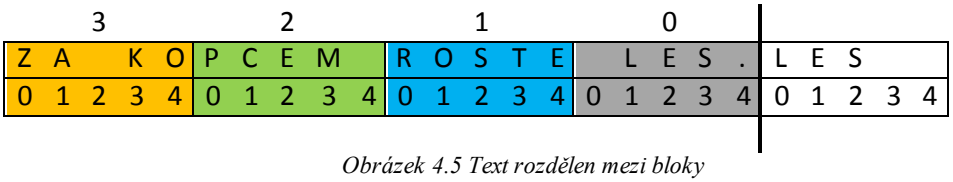
Obrázek 4.4 Znázornění rozdělení bloků nad textem

Samotné vyhledávání je tedy pouze v rámci jednoho bloku. Existuje však jedna výjimka. Tato výjimka nastává v situaci, kdy se hledaný text nachází na konci bloku. Jelikož jsou vyhledávaná data v globální paměti, může klidně shoda z konce jednoho bloku zasáhnout až na začátek bloku dalšího. Obrázek 2.3 přesně zobrazuje tuto situaci.

Každé vlákno v bloku má přidělen svůj jedinečný identifikátor, díky kterému jsme schopni vlákno rozmístit na každý znak v textu. Stejně tak má každý blok svůj jedinečný identifikátor. Obrázek 4.5 zobrazuje, jak vypadá rozmístění bloků nad textem. Pro dobré pochopení je zobrazena situace, kdy se hledá ve 20 znacích. Celá tato operace probíhá najednou, tedy paralelně. Text, který hledáme je zobrazen na konci v bílém bloku. Černá dělicí čára mezi bloky značí předěl mezi zakódovanou a nezakódovanou částí okénka. Jednotlivá čísla pod textem značí index vlákna a čísla nad bloky značí index bloku. Hledání na grafické kartě je tedy spuštěno ve 4 blocích po 5 vláknech. Všechny bloky jsou tedy spuštěny paralelně najednou.

Každé vlákno v každém bloku porovná písmeno dle svého indexu s prvním písmenem hledaného textu.
Zkratka pro blok je B, pro vlákno V. Porovnání vypadá takto:

B3, V0: Z != L	B2, V0: P != L	B1, V0: R != L	B0, V0: _ != L
B3, V1: A != L	B2, V1: C != L	B1, V1: O != L	B0, V1: L = L
B3, V2: _ != L	B2, V2: E != L	B1, V2: S != L	B0, V2: E != L
B3, V3: K != L	B2, V3: M != L	B1, V3: T != L	B0, V3: S != L
B3, V4: O != L	B2, V4: _!= L	B1, V4: E != L	B0, V4: .!= L



Obrázek 4.5 Text rozdělen mezi bloky

Všechna vlákna v bloku mají vyčleněnou sdílenou paměť pro výsledky. Každé vlákno v kernelu má vyčleněn jeden registr, do kterého zapíše délku hledané shody. Není-li nalezena shoda ani jednoho znaku, uloží se do výsledků hodnota 0,0,znak. Pokud vlákno nalezne shodu, pak se posune jak v rámci bloku, tak i v hledaném text o jeden znak vpravo. Tato situace nastala pouze pro vlákno číslo 1 z bloku 0. Posun se opakuje do té doby, dokud existuje shoda mezi znaky. Počet shod se zaznamenává pro každé vlákno do registru. Při první neshodě znaků, se pouze vyhodnotí hodnota uložená v registru, a není-li nulová, je do výsledného pole uložena délka shody, spočítaný offset, což je vlastně index vlákna + index bloku*počet vláken a další znak za shodou.

Jelikož je počet vláken v bloku 256, tak počet možných shod je 256. Po skončení samotného prohledávání nastává další fáze a to nalezení maximální možné shody. K tomu je možné přistoupit dvěma způsoby. Zpracovat v rámci každého bloku výsledky sekvenčně přímo v kernelu, nebo použitím paralelní redukce.

Paralelní redukce [5] je způsob, jak paralelně v poli najít maximum, minimum či spočítat součet všech hodnot. Mějme pole o 256 prvcích. Prohledávání funguje následovně. V každém kroku spustíme polovinu počtu vláken ještě neprohledaných dat, kdy každé vlákno porovná prvek dle svého indexu + 128. Je-li hodnota vpravo větší, než hodnota vlevo, hodnoty přehodíme. Poté opět spouštíme stejný proces znovu, ale nyní už pro 128 hodnot s 64 vlákny. Proces se opakuje, dokud není na pozici 0 v poli maximum. Počet opakování je tedy 7.

Obrázek 4.6 zobrazuje proces paralelní redukce. Pro jednoduchost je zde zobrazeno pole o 8 hodnotách, *N* značí záporné číslo, tedy nulovou délku shody. Stejná barva značí prvky, které jsou spolu porovnávány. Sudé řádky zobrazují data, po provedení redukčního kroku.

1	N	5	2	1	N	N	9
1	N	5	9	1	N	N	9
1	N	5	9	1	N	N	9
5	9	5	9	1	N	N	9
5	9	9	9	1	N	N	9
9	9	5	9	1	N	N	9
9	9	5	9	1	N	N	9

Obrázek 4.6 Paralelní redukce

Výsledek paralelní redukce, tedy maximální hodnota se uloží do globální paměti pod indexem bloku, v kterém byla tato hodnota nalezena. Tato hodnota reprezentuje maximální shodu, avšak v rámci jednoho bloku. Abychom našli celkové maximum, je znovu zapotřebí porovnat všechny maximální hodnoty bloků a vybrat tu největší. Data s výsledky se tedy překopírují zpět do hosta, a opět se hledá maximální hodnota ze všech bloků. Bude-li velikost vyhledávacího okénka 4 miliony znaků, bude spuštěno 15625 bloků, což znamená, že velikost pole bude taktéž 15625. Pro rychlé nalezení maximální hodnoty jsem chtěl využít strukturu binárního stromu, jenže jak se ukázalo, vytvoření tohoto stromu by trvalo déle, než je samotné sekvenční prohledání takto získaných dat. Po nalezení absolutního maxima, jsou zapsána data na výstup. Poté se algoritmus posune v textu dále o počet znaků rovný délce nalezené shody. Poté se proces hledání opakuje znovu.

Tento algoritmus jsem se následně pokusil urychlit lepším využitím sdílené paměti. Data, která se porovnávají jsem nejprve překopíroval do sdílené paměti a teprve v ní se samotné porovnávání odehrávalo. Bohužel se mi nikdy nepovedlo dosáhnout lepších výsledků s takto využitou sdílenou pamětí. Proto jsem sdílenou paměť pro zdrojová data nepoužil.

4.4.4. Optimalizovaná verze s dvojí paralelní redukcí

Nalezení maximální délky shody v datech zpracovaných grafickou kartou zabíralo relativně velké množství času. Důvodem je sekvenční procházení pole na procesoru. Tento krok lze však taktéž efektivně paralelizovat. Máme-li okénko o velikosti 4M, tak počet možných trojic v kterých musíme najít maximální hodnotu je 16384. Toto pole však lze znovu efektivně zpracovat pomocí paralelní redukce. Aplikuje se tedy znovu stejný postup jako na nalezená data v rámci kernelu. S tím rozdílem že počet spuštěných bloků je nyní 16384/256. Výsledkem je tedy pole o 64 prvcích. Prohledat toto pole sekvenčně již není tak zdlouhavé, jako tomu bylo u pole o 16384 prvcích. Určitě vás napadne, proč není znovu provedena paralelní redukce na těchto 64 hodnot. Jelikož by na prohledání takto velkého pole stačil pouze jeden blok, je neefektivní spouštět redukci na GPU, lepších výsledků bylo dosaženo při sekvenčním prohledání tohoto pole.

Pro správnou funkci paralelní redukce je nutné, aby počet spuštěných bloků byl beze zbytku dělitelný počtem jader. Tedy okénko o velikosti Nutno podotknout, že tato verze v sobě skrývá malou chybu. Ačkoliv jsem zdrojový kód pečlivě prohledal, zdroj chyby jsem nenašel. Chyba spočívá v tom, že v průměru jedna až pět trojic z 500000 je špatně zakódovaná. Samostatný zdrojový kód redukce není až tak složitý, avšak nemožnost ladit kód na jedné grafické kartě mě omezila na ladění pomocí výpisů, které není efektivní a hledání chyb je tudíž nadlidský úkol.

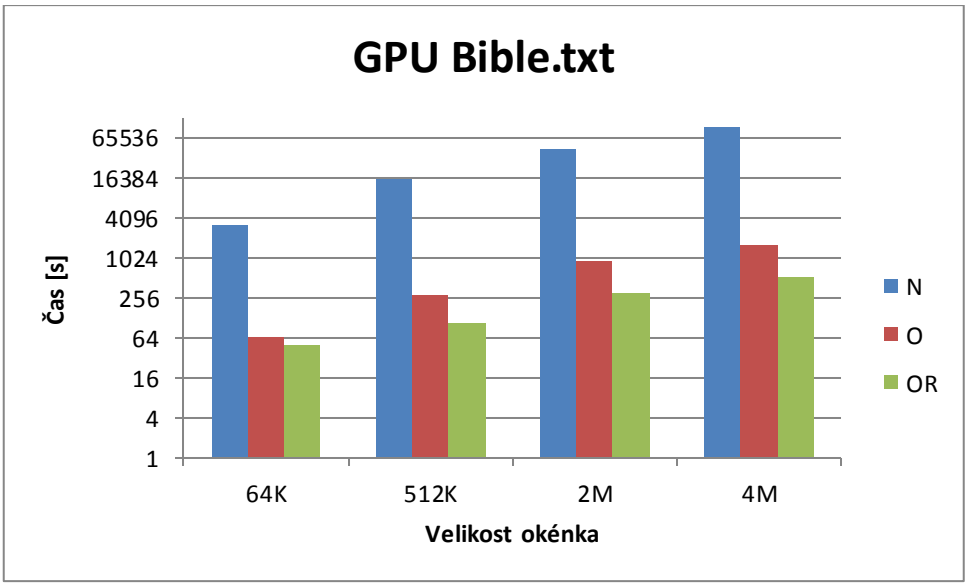
4.5. Srovnání GPU verzí LZ77

GPU verzi LZ77 jsem testoval na GPU nVidia GeForce 555M (96 CUDA jader). Karta obsahuje 2 multiprocesory po 48 jádrech. Nejedná se tedy o žádnou profesionální výpočetní kartu, tomu také odpovídají i výsledky.

Získané výsledky byly naměřeny na těchto verzích:

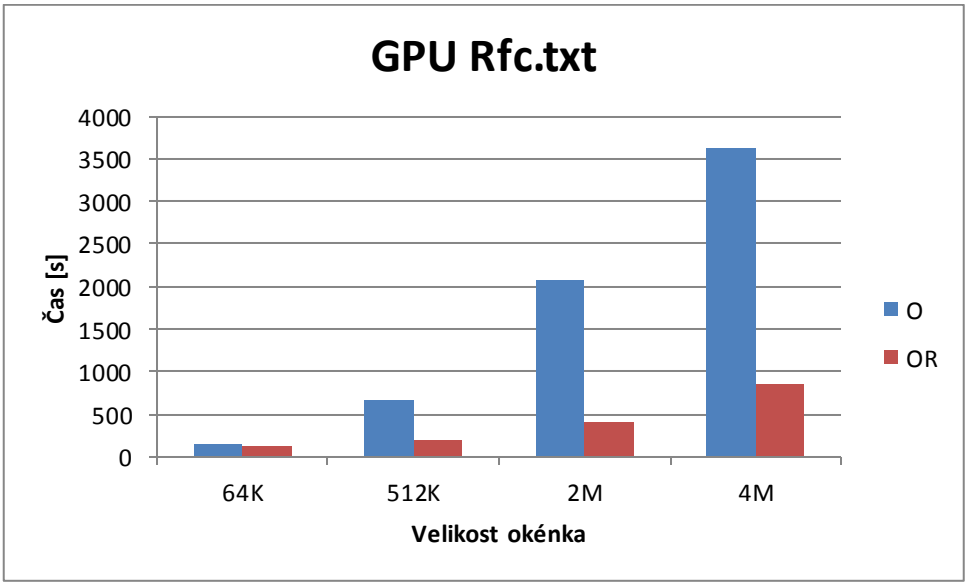
- Naivní (N)
- Optimalizovaná (O)
- Optimalizovaná s dvojí paralelní redukcí (OR)

Naivní verze byla testována pouze pro text bible.txt z důvodů velké časové náročnosti. Masově paralelní verze byla z testů vyřazena. Důvodem bylo opravdu nadměrné zatížení GPU. Nemá smysl testovat každou verzi jako takovou. Důležitým faktem je, že díky nedostatkům předchozích verzí, jsem byl schopen se zaměřit na neefektivní části kódu a ty v dalších verzích odstranit.



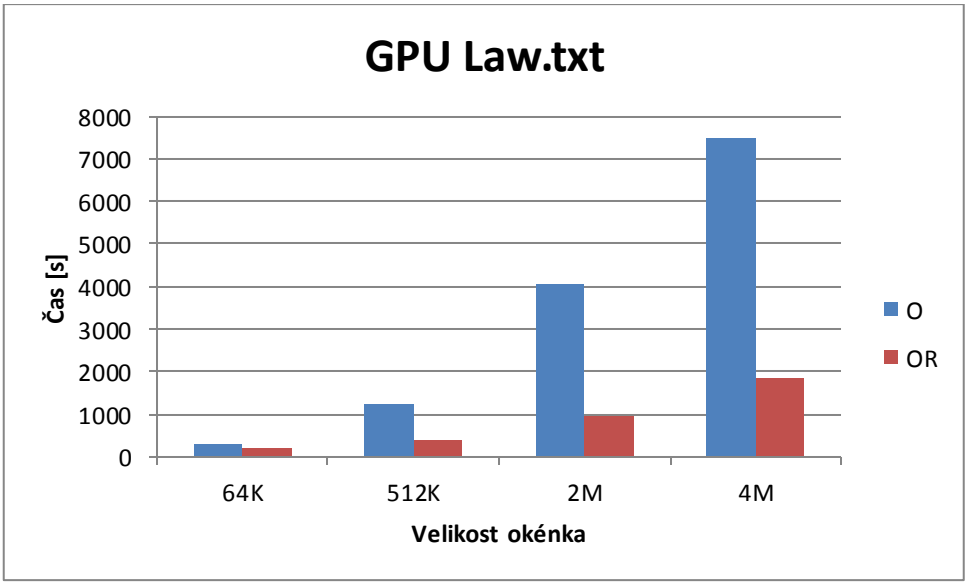
Obrázek 4.7 Rychlost GPU algoritmů nad souborem bible.txt

Pro časovou osu jsem zvolil v tomto případě logaritmické měřítko o základu 2. Čas komprimace N verze byl ve srovnání s OR mnohokrát větší, takže by při klasickém zobrazení zanikla informace o časové náročnosti OR algoritmu. V posledním případě byl čas pro zakódování pomocí N verze více jak jeden den! Verze OR zkomprimovala soubor za necelých 9 minut. Rychlost OR verze je v tomto případě tedy 180x vyšší než rychlost verze naivní.



Obrázek 4.8 Rychlost GPU algoritmů nad souborem rfc.txt

Rychlost OR verze zvětšuje podle toho, jak velká oblast je potřeba prohledat. Nemá smysl vykonávat paralelní redukci na GPU při relativně malém počtu prvků. Za stejnou dobu totiž procesor zvládne tyto výsledky zpracovat ve stejném časovém období. Při použití 4M okénka je již ale počet analyzovaných prvků 16384, takže se rozhodně vyplatí spustit prohledávání na GPU. Rychlost OR verze je v tomto případě 4x větší než u verze O.



Obrázek 4.9 Rychlost GPU algoritmů nad souborem law.txt

Zde je vidět, že dvojitá paralelní redukce zrychlí čas potřebný ke komprimaci souboru až při použití nezakódované části větší než 512k. Čas OR je oproti O tím kratší, čím větší je velikost okénka. V posledním případě je časová složitost OR verze 4x menší než je tomu u verze O.

5. Srovnání GPU a CPU verze

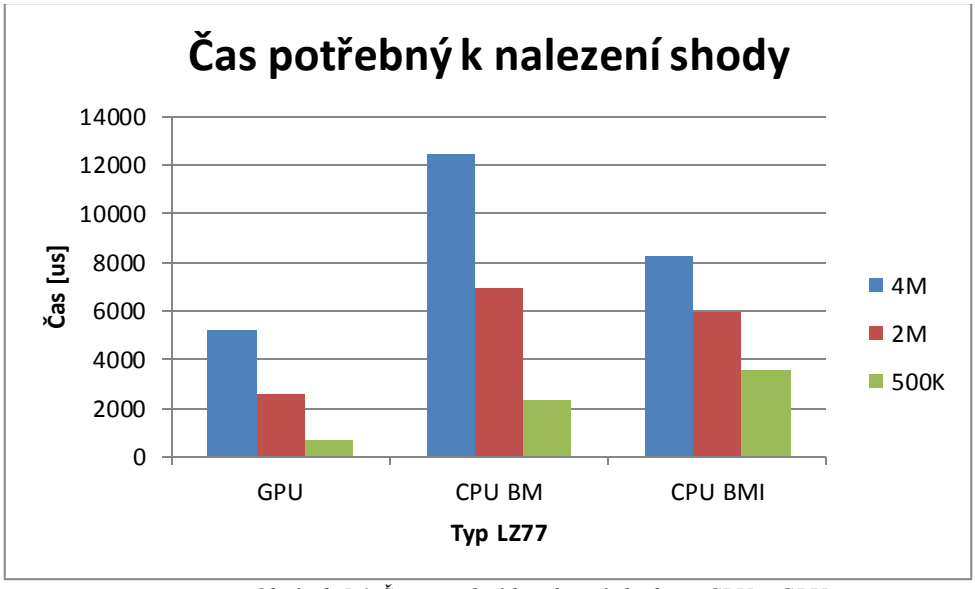
Jak CPU tak GPU verze mají své pro a proti. Největší kámen úrazu sekvenčního zpracování je zcela jistě samotné vyhledávání shod, u paralelní verze to může být následné zpracování již prohledaných dat.

5.1. Hledání vzorku v textu

Důvodem implementace LZ77 na grafické kartě bylo prozkoumat způsoby, jak efektivně vyhledávat shody v textu a urychlit tak tento časově náročný problém. V grafu jsou zobrazeny výsledky, pokud bychom hledali jednu shodu v okénku o velikosti 500K, 2M a 4M. Počet iterací k nalezení shody jsem zvolil 20, abych vyloučil možnost náhodného jevu.

Získané výsledky byly naměřeny na těchto verzích:

- Optimalizovaná GPU
- CPU B-M bez využití informace (BM)
- CPU B-M s využitím informace (BMI)



Obrázek 5.1 Čas potřebný k nalezení shody na CPU a GPU

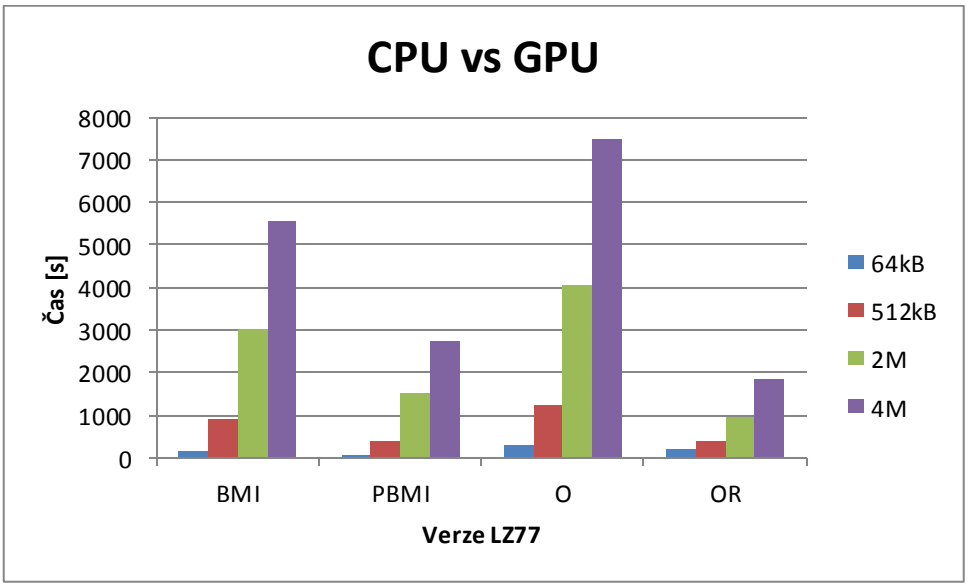
Jak lze vidět, čas potřebný k nalezení shody na grafické kartě je výrazně nižší než čas pro nalezení shody na procesoru. Důvodem je způsob vyhledávání grafické karty, kdy je schopna spustit hledání nad textem po částech paralelně, oproti sekvenčnímu hledání procesoru. Zrychlení v posledním případě je cca. 5,4x. S větším počtem multiprocesorů by grafická karta mohla dosáhnout ještě lepších výsledků. Důvodem by bylo více paralelně spuštěných bloků pro hledání shody.

5.2. Kódování LZ77

V grafech je srovnána rychlost CPU a GPU verze LZ77. Textem byla opět bible.txt, rfc.txt a law.txt s velikostmi 4MB, 10MB, 20MB. Velikost okének byla 4M znaků.

Získané výsledky byly naměřeny na těchto verzích:

- Optimalizovaná GPU (O)
- Optimalizovaná GPU s dvojí paralelní redukcí (OR)
- CPU B-M s využitím informace (BMI)
- CPU Paralelní B-M s využitím informace (PBMI)



Obrázek 5.2 Porovnání rychlostí CPU a GPU algoritmů

Zde již GPU verze není tolikrát rychlejší, jako tomu bylo u vyhledávání jednoho vzorku v textu. Důvodů je hned několik. GPU je schopno produkovat v relativně velmi krátkém čase mnoho dat, které pak musí procesor analyzovat. Při velikosti vyhledávacího okénka 4M je počet bloků 15625. Procesor proto musí prohledat pole o velikosti 15625 prvků. Množství těchto dat je zredukováno v případě druhé GPU verze, kdy je zahrnuta dvojí paralelní redukce. Tedy GPU samo zpracuje tyto výsledky, tudíž procesoru pak stačí v případě 4M okénka prohledat pouze pole o velikosti 60 prvků. I přes tyto všechny problémy je však GPU OR verze 3x méně časově náročná než BMI CPU verze.

6. Závěr

Ačkoliv se z grafů může zdát, že GPU verze LZ77 zaostává, nemusí tomu tak být. Porovnání obou verzí bylo prováděno na CPU Intel Core i7 2670QM, což je v dnešní době v noteboocích high-endový procesor, CPU Intel Core 2 Duo E8400 a GPU nVidia GeForce 555M (96 Cuda jader). Výkon karty je oproti špičkovým výpočetním kartám jako je nVidia Tesla C2050 s 448 Cuda jádry relativně velmi malý. Použitím silnější GPU jakou je Tesla C2050, GeForce GTX 560 či GTX 580 by mohlo vést k daleko lepším výsledkům.

Co se této práce týče, musím podotknout, že díky tomuto tématu jsem se naučil mnoho nového, hlavně z oblasti komprese textu. Komprese je určitě zajímavé odvětví a právě paralelismus by mohl vést k urychlení celého procesu. Mnoho znalostí jsem také načerpal z oblasti architektury samotné grafické karty a to konkrétně nVidia Fermi. Samotná práce mě nadchla, bohužel nemožnost ladit kód na jedné grafické kartě se mi mnohokrát vymstila. Hledání chyb ve zdrojovém kódu části grafické karty s mnoha indexy nebylo vůbec jednoduché.

Spousta laiků si myslí, že platí přímá úměrnost mezi počtem jader a výkonem čipu. To ale vůbec není pravda. Díky naprosto jiné architektuře paralelních čipů a odlišnému způsobu programování je výkon velmi závislý na dané úloze. Jak již bylo začátkem práce uvedeno, masivně paralelní čipy jsou vhodné pro zpracování obrazu, kdy je obraz rozdělen do jednotlivých částí, které mohou být provedeny najednou. Typickým příkladem je převedení obrazu do odstínu šedi, kdy stačí pouze změnit hodnotu RGB složky každého pixelu. Přesně takové typy úloh jsou pro paralelní zpracování nejlepší, tedy naprosto **stejná množina operací** bez složitějších podmínek nad velkým množstvím dat. To ale u LZ77 není možné. Data je nutno porovnávat, tudíž každé vlákno v rámci bloku provádí jiný počet operací v závislosti na délce shody. To způsobuje rozchod vláken v rámci warpu a následnou serializaci výpočtu. Celé vyhledávání je poté daleko pomalejší. Dalším problémem je následné zpracování výsledků, které avšak lze urychlit díky paralelní redukci.

Nejtěžší na celé věci bylo vymyslet způsob, jak bude samotné paralelní hledání fungovat. Není vyloučeno, že bude existovat efektivnější způsob jak celou věc naprogramovat. Nejednou se mi také stalo, že skvělá myšlenka následně ztroskotala na maličkosti, která nebyla na začátku až tak zřejmá. Případné další vylepšení algoritmu by pak mohlo plynout z lepší znalosti architektury GPU, či naprosto jiné organizace dat.

7. Reference

[1] SALOMON, David. *Data Compression*: *the complete reference*. 4th ed. London: Springer, 2007, 1092 s. ISBN 978-1-84628-602-5.

[2] Referát o vyhledávání v textu. BENEŠ, Miroslav. *Vyhledávání v textu* [online]. 2001 [cit. 2011-10-14]. Dostupné z: http://htmltolatex.sourceforge.net/samples/sample3.html

[3] NVIDIA CUDA C Programming Guide. NVIDIA. *Developer.nvidia.com* [online]. 16.4.2012 [cit. 2011-11-07]. Dostupné z: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[4] NVIDIA CUDA C BEST PRACTICES GUIDE. NVIDIA. *Developer.nvidia.com* [online]. 11.1.2012 [cit. 2011-11-14]. Dostupné z: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf

[5] Parallel reduction. http://*www.cse.nd.edu* [online]. 12.8.2008 [cit. 2012-04-11]. Dostupné z: http://www.cse.nd.edu/courses/cse40833/www/kogge_gpu/lectures/logsum.pdf